Figure 1

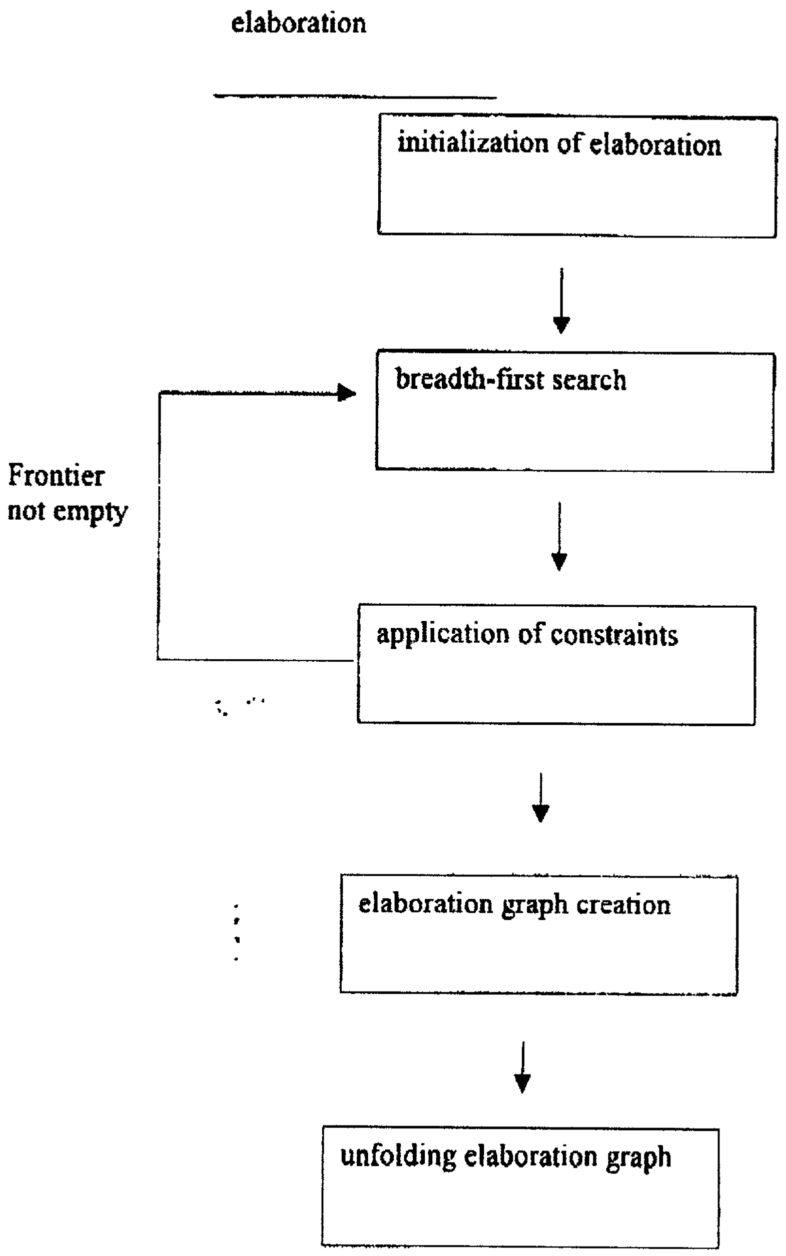

elaboration
initialization of elaboration
breadth-first search
Frontier not empty
application of constraints
elaboration graph creation
unfolding elaboration graph

Figure 1 (con't)

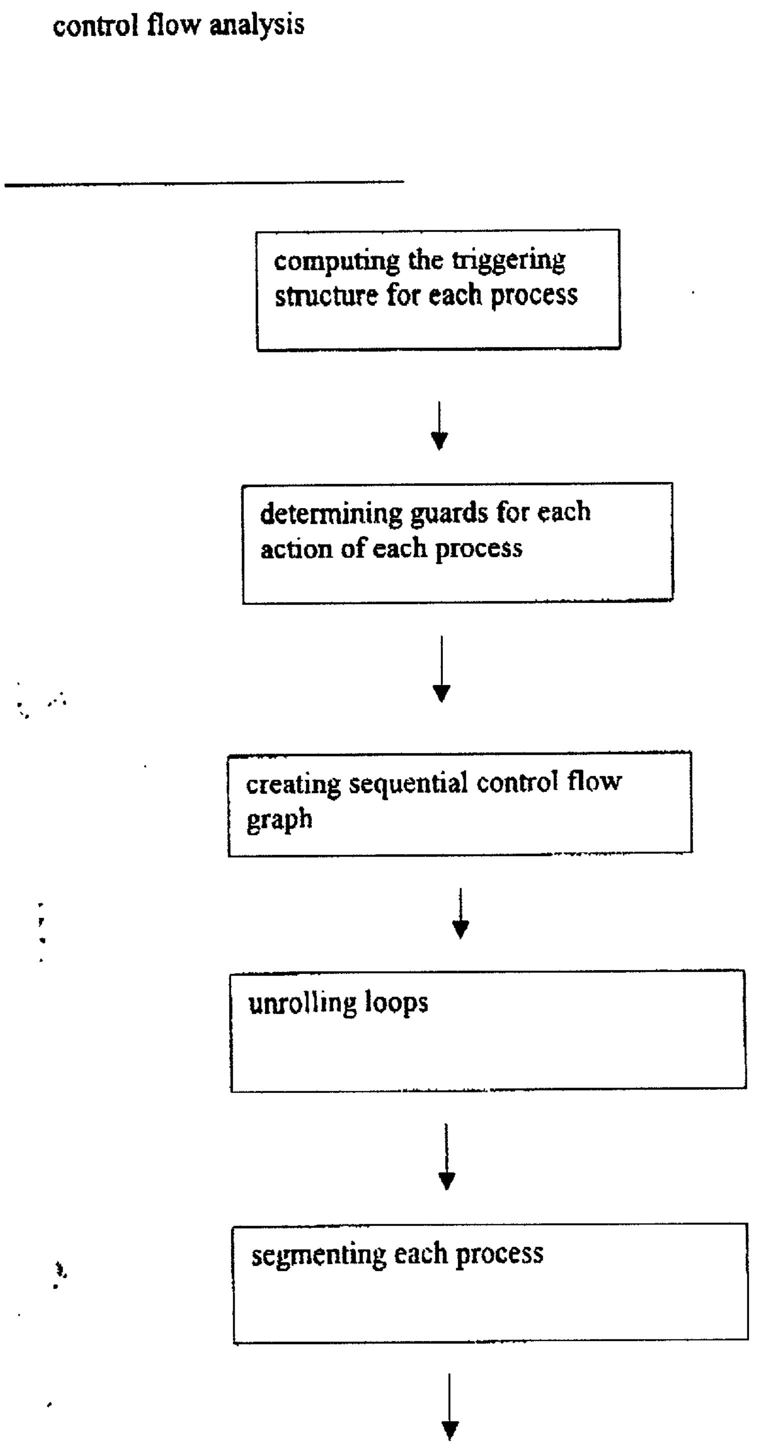

control flow analysis
computing the triggering structure for each process
determining guards for each action of each process
creating sequential control flow graph
unrolling loops
segmenting each process

Figure 1 (con't2)

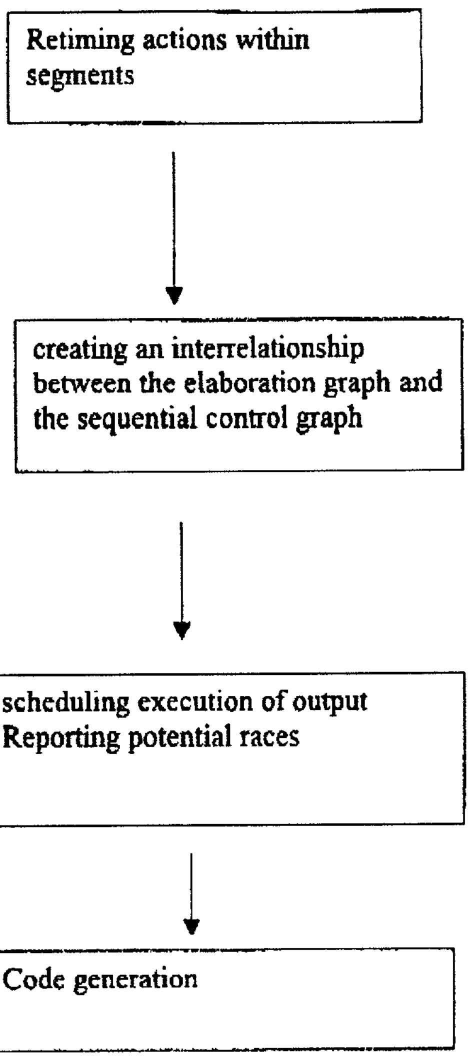

Retiming actions within segments
creating an interrelationship between the elaboration graph and the sequential control graph
scheduling execution of output Reporting potential races
Code generation

Figure 2: Elaboration Algorithm

```
L ← refto( top unit )
C = Φ
while L != Φ
   for each reference r in L begin
      NL ← Φ
      t ← typeof( r )
      r.target ← makenode( t )
      C ← C + {constraints of t}
      for each field f in t begin
         NL ← NL + refto( f )
         C ← apply( C )
      end
   end
   L ← L + NL
end
```

Figure 3

```
<'

struct cl {                              // arbiter client
    id          :int;                     // my id
    data        :int;                     // data - INPUT
    !drdy       :bool;                    // data ready - INPUT
    !xreq       :bool;                    // transfer request - interface to arb
    !xgrt       :bool;                    // transfer grant - arb sets this
    arb         :arb;
    keep arb == sys.arb;
};

struct arb {
    cls         :list of cl;
    data        :int;                     // data destination
};

extend sys {
    cl_list     :list of cl;
    keep cl_list.size() == 4;
    keep for each in cl_list {
        .id == index;
    };
    arb         :arb;
    keep arb.cls == cl_list;
};

'>
```

Figure 4

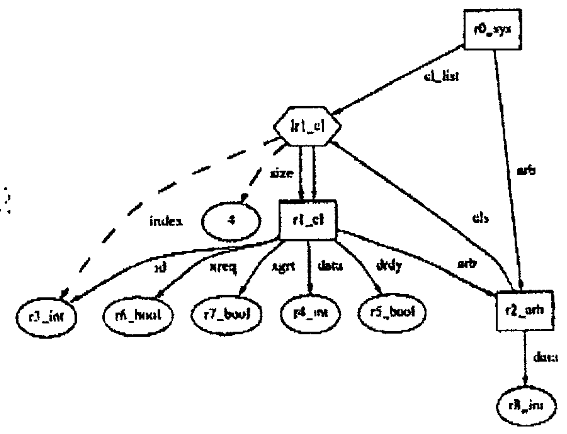

r0_sys
cl_list
arb
lr1_cl
size
index
4
r1_cl
cls
id
xreq
xgrt
data
drdy
arb
r3_int
r6_bool
r7_bool
r4_int
r5_bool
r2_arb
data
r8_int

Figure 5A

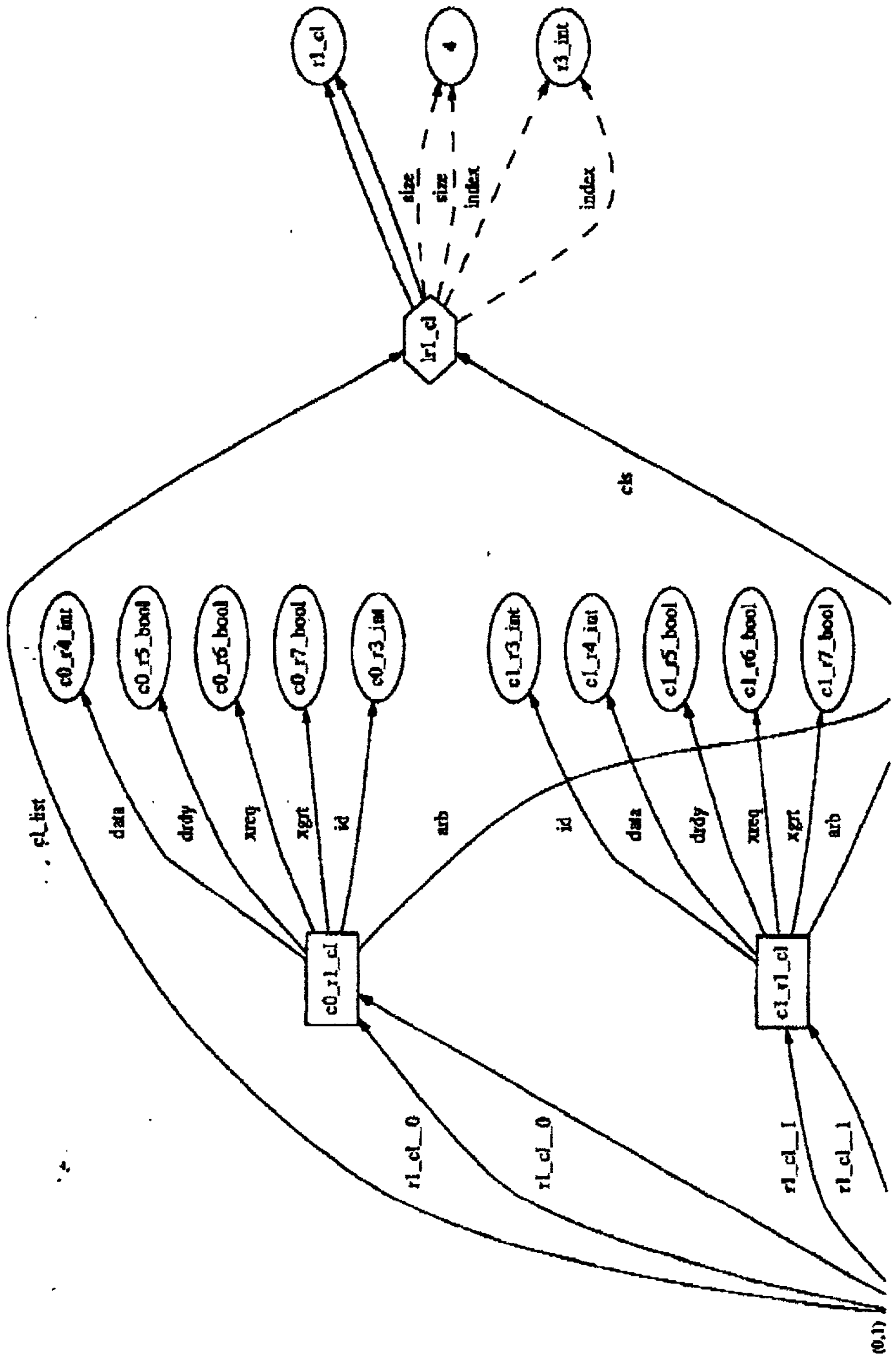
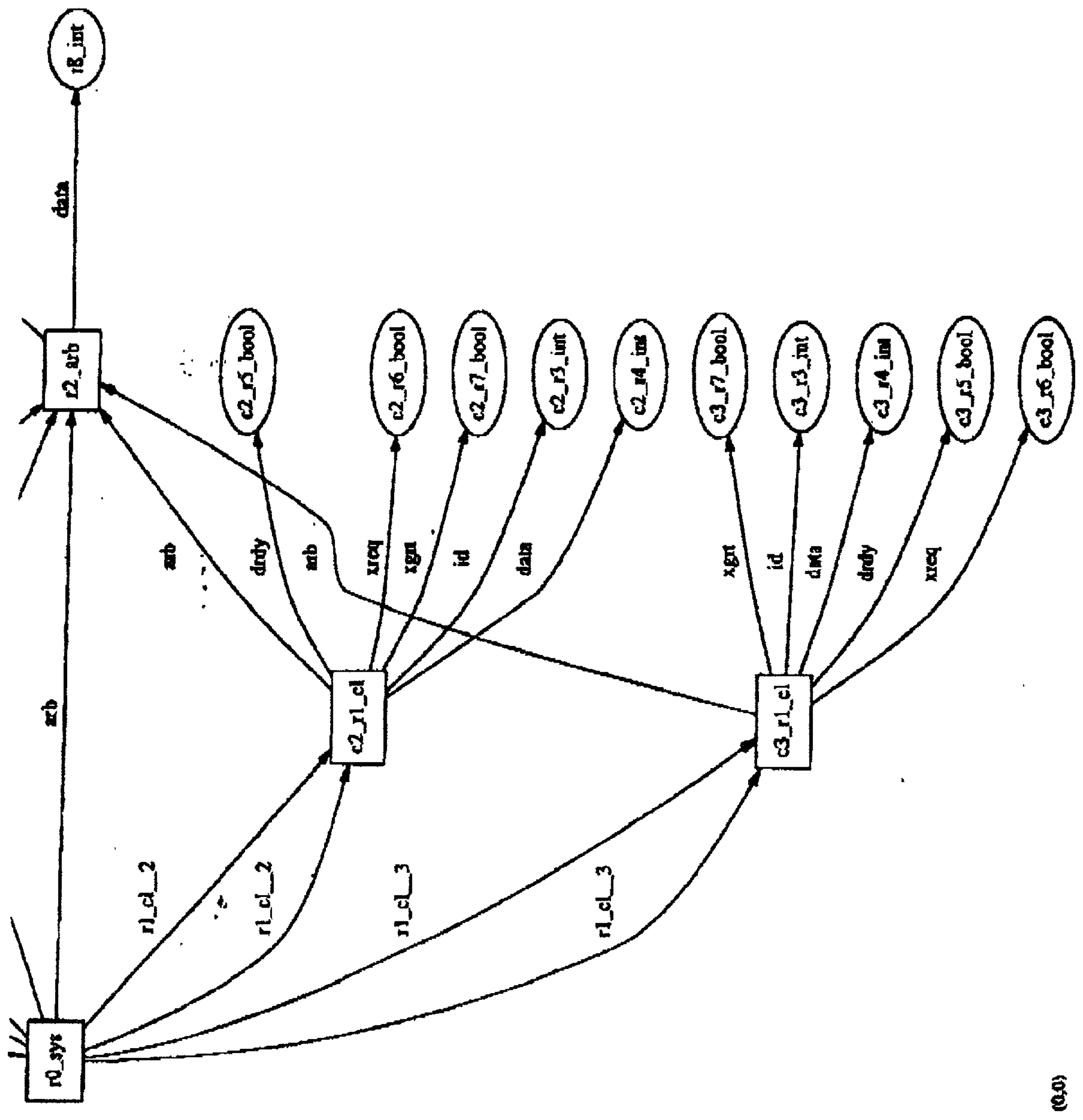

r1_cl
4
r3_int
size
size
index
index
lr1_cl
cls
cl_list
c0_r4_int
c0_r5_bool
c0_r6_bool
c0_r7_bool
c0_r3_int
data
drdy
xreq
xgrt
id
arb
c1_r3_int
c1_r4_int
c1_r5_bool
c1_r6_bool
c1_r7_bool
id
data
drdy
xreq
xgrt
arb
c0_r1_cl
c1_r1_cl
r1_cl__0
r1_cl__0
r1_cl__1
r1_cl__1
(0,1)

Figure 5B


r0_sys
r2_arb
r8_int
data
arb
r1_cl_2
r1_cl_2
r1_cl_3
r1_cl_3
c2_r1_cl
c3_r1_cl
arb
drdy
arb
xreq
xgnt
id
data
xgnt
id
data
drdy
xreq
c2_r5_bool
c2_r6_bool
c2_r7_bool
c2_r3_int
c2_r4_int
c3_r7_bool
c3_r3_int
c3_r4_int
c3_r5_bool
c3_r6_bool
(0,0)

```
<'
struct cl {                              // arbiter client
    id          :int;                    // my id
    data        :int;                    // data - INPUT
    !drdy        :bool;                   // data ready - INPUT
    !xreq        :bool;                   // transfer request
    !xgrt        :bool;                   // transfer grant
    arb         :arb;
    keep arb == sys.arb;

    trans() @sys.clk is {
        while TRUE {
            wait true(drdy);
            xreq = TRUE;
            wait true(xgrt);
            arb.data = data;
            wait cycle;
            xreq = FALSE;
            wait true(not xgrt);
            drdy = FALSE;
        };
    };
};

struct arb {
    cls         :list of cl;
    data        :int;                // data destination
    switch() @sys.clk is {
        while TRUE {
            for each in cls {
                if .xreq then {
                    .xgrt = TRUE;
                    wait true(not .xreq);
                    .xgrt = FALSE;
                };
            };
            wait cycle;
        };
    };
};

extend sys {
    cl_list     :list of cl;
    keep cl_list.size() == 4;
    keep for each in cl_list {
        .id == index;
    };
    arb         :arb;
    keep arb.cls == cl_list;
    event clk;
};
'>
```

Figure 6

Figure 7

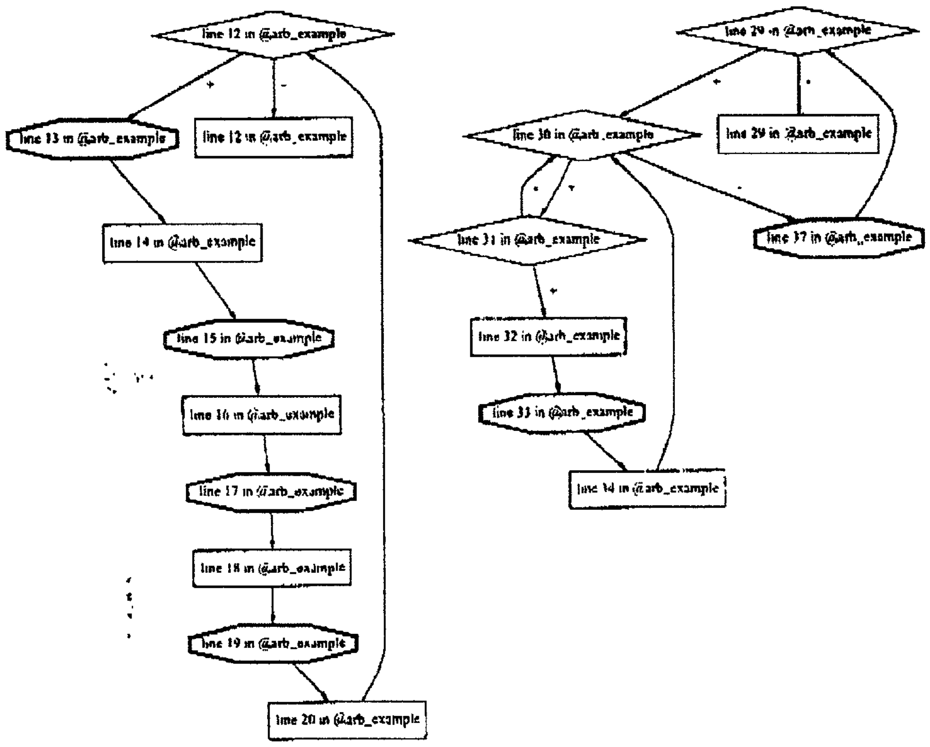
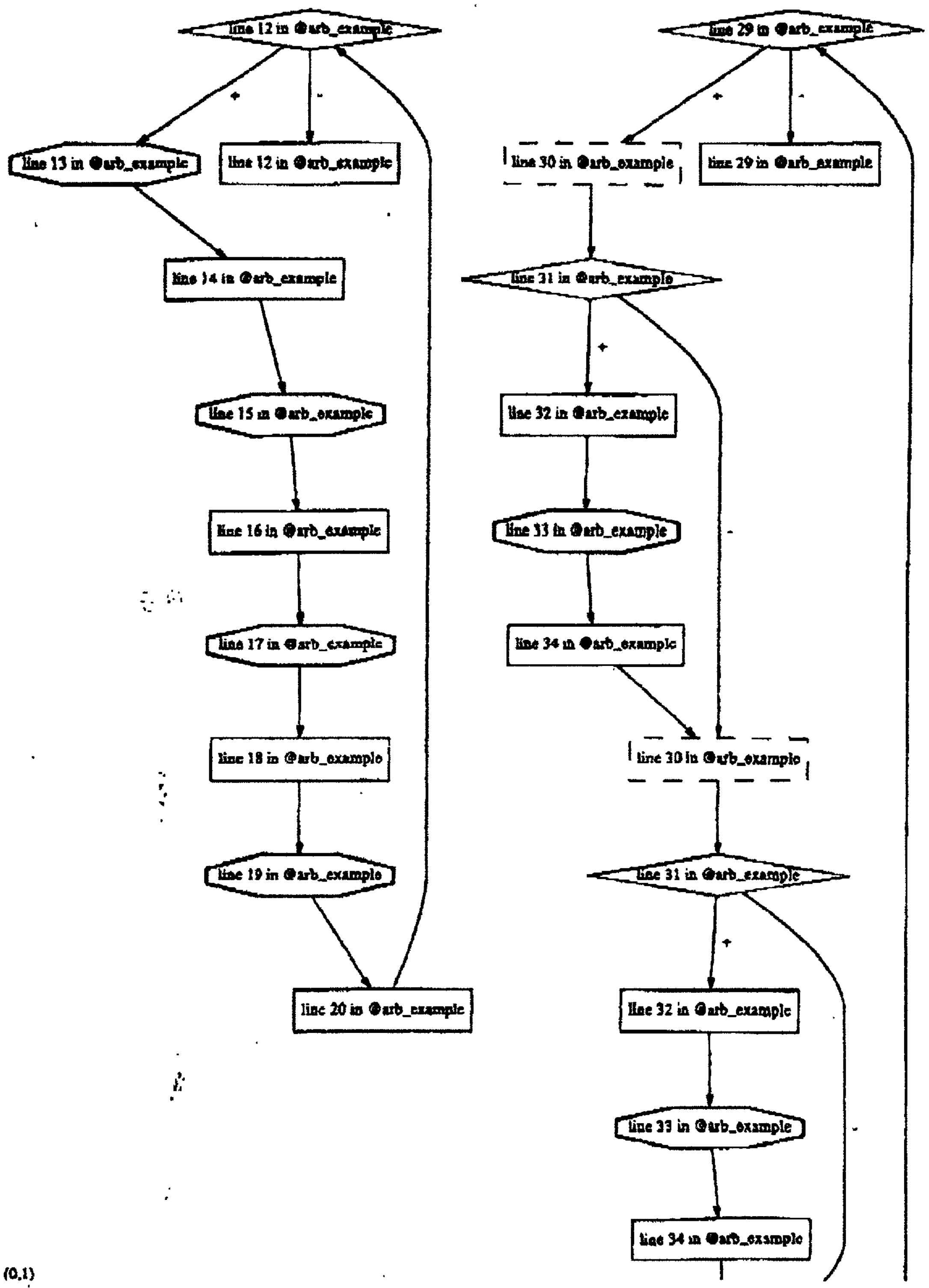
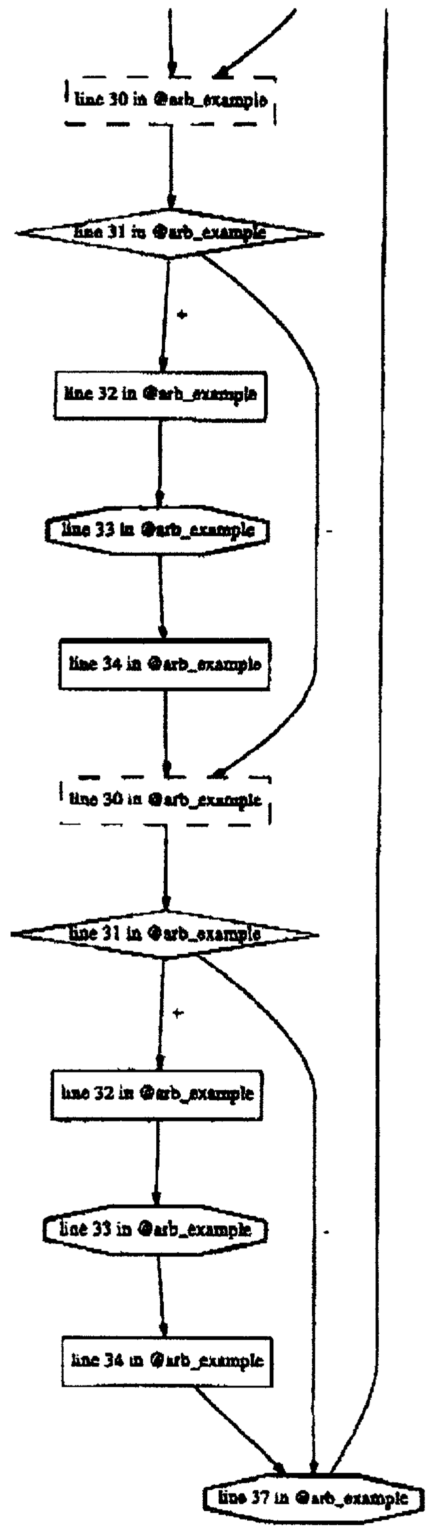

line 12 in @arb_example
+
-
line 13 in @arb_example
line 12 in @arb_example
line 14 in @arb_example
line 15 in @arb_example
line 16 in @arb_example
line 17 in @arb_example
line 18 in @arb_example
line 19 in @arb_example
line 20 in @arb_example
line 29 in @arb_example
+
-
line 30 in @arb_example
line 29 in @arb_example
-
+
-
line 31 in @arb_example
line 37 in @arb_example
+
line 32 in @arb_example
line 33 in @arb_example
line 34 in @arb_example

Figure 8 Part I



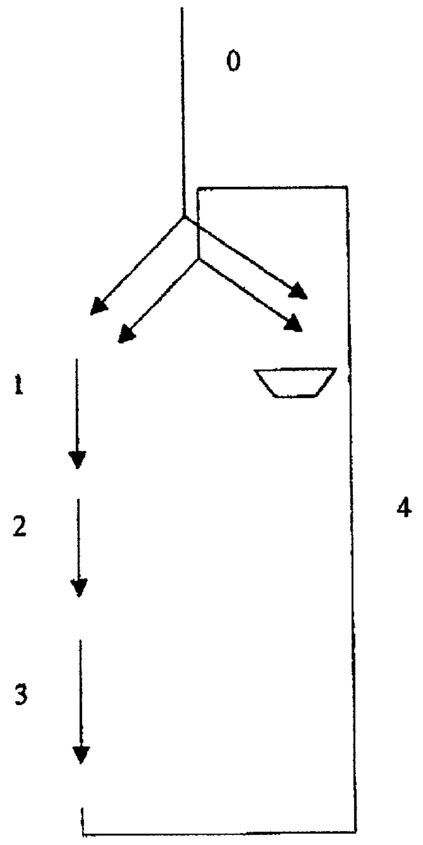
(0,1)

Figure 8 Part II



(0,0)

Figure 9: Segmentation of a control flow graph



```
for each node n in EG such that n has processes begin
  for each process p in n begin
    for each segment s in p begin
      for each action a in s begin
        for each read expression e in a begin
          t ← evaluate( e, context )
          tag t with { n, s, 'read' }
        end
        for each write expression e in a begin
          t ← evaluate( e, context )
          tag t with { n, s, 'write' }
        end
      end
    end
  end
end
```

Figure 10

```
Peterson's mutex algorithm - simple two agent example

<'

struct agent {
    req     :bool;
    id      :int;
    oa      :agent;
    p() @sys.clk is {
        req = TRUE;
        sys.k = id;
        while (sys.k == id) && oa.req {
            wait cycle;
        };
        wait cycle;
        sys.w = id;     // Critical segment
        req = FALSE;
    };
};

extend sys {
    k   :int;   // Requestors id.
    w   :int;   // The protected data field
    a0  :agent;
    a1  :agent;
    keep a0.id == 0;
    keep a0.oa == a1;
    keep a1.id == 1;
    keep a1.oa == a0;
    event clk;
};

'>
```

Figure 11

Figure 12

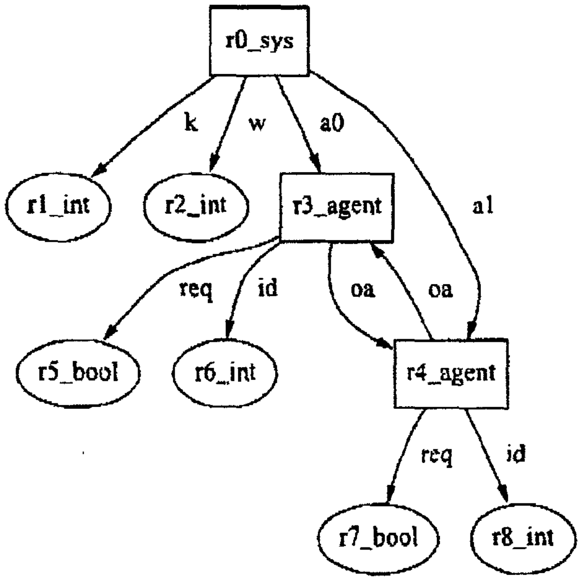

r0_sys
k
w
a0
a1
r1_int
r2_int
r3_agent
req
id
oa
oa
r5_bool
r6_int
r4_agent
req
id
r7_bool
r8_int

Figure 13

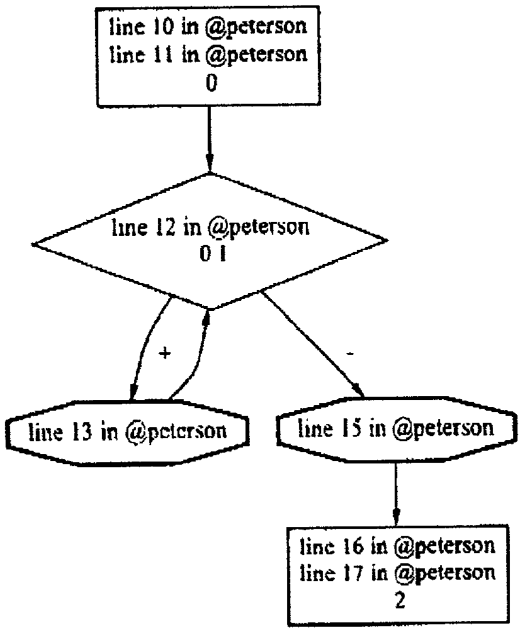
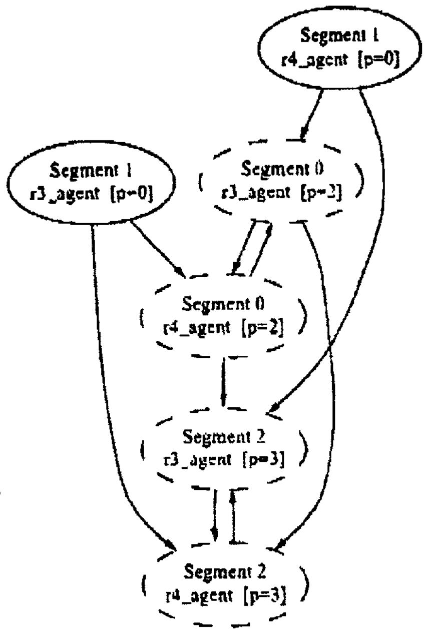

line 10 in @peterson
line 11 in @peterson
0
line 12 in @peterson
0 1
+
-
line 13 in @peterson
line 15 in @peterson
line 16 in @peterson
line 17 in @peterson
2

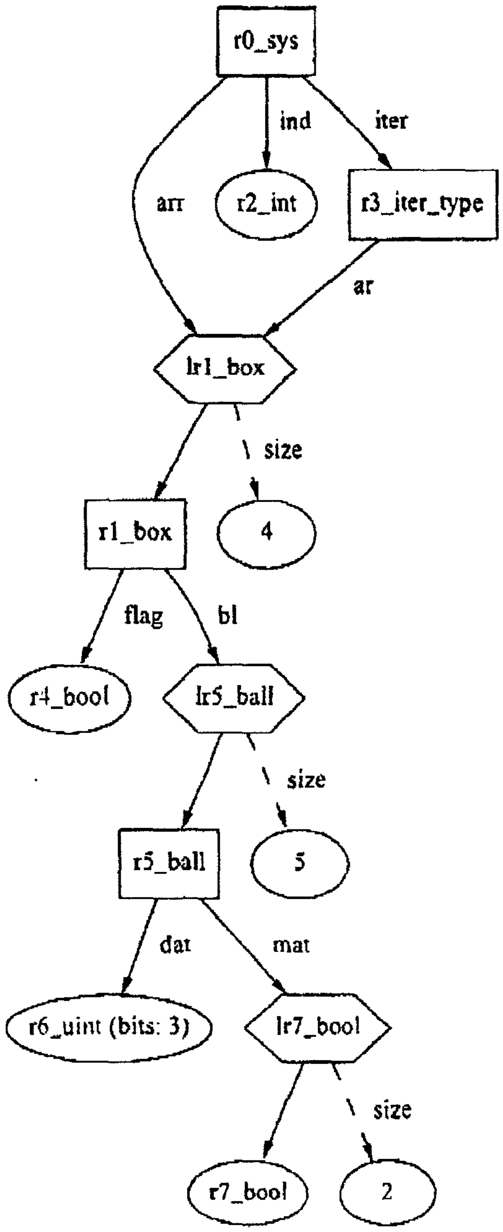
Figure 14


Segment 1
r4_agent [p=0]
Segment 1
r3_agent [p=0]
Segment 0
r3_agent [p=2]
Segment 0
r4_agent [p=2]
Segment 2
r3_agent [p=3]
Segment 2
r4_agent [p=3]

```
This is an iterator access to hierarchical arrays

<'
struct ball {
    dat     :uint (bits:3);
    mat     :list of bool;
    keep mat.size() == 2;
};

struct box {
    flag    :bool;
    bl     :list of ball;
    keep bl.size() == 5;
};

struct iter_type {
    ar      :list of box;
    foo() @sys.clk is {
        wait cycle;
        for each in ar {
            .flag = TRUE;
            for each in .bl {
                .dat = 2;
                .mat[1] = FALSE;
            };
        };
        ar[2].bl[3].mat[0] = TRUE;
        ar[2].bl[sys.ind].mat[0] = TRUE;
    };
};

extend sys {
    event clk;
    arr     :list of box;
    keep arr.size() == 4;
    ind     :int;

    iter    :iter_type;
    keep iter.ar == arr;
};

'>
```

Figure 15

Figure 16


r0_sys
arr
ind
iter
r2_int
r3_iter_type
ar
lr1_box
\ size
r1_box
4
flag
bl
r4_bool
lr5_ball
\ size
r5_ball
5
dat
mat
r6_uint (bits: 3)
lr7_bool
\ size
r7_bool
2